SimClusters: Community-Based Representations for Heterogeneous Recommendations at Twitter

Venu Satuluri, Yao Wu
Twitter, Inc.
{vsatuluri, yaow}@twitter.com

Xun Zheng*
Carnegie Mellon University
xunzheng@cs.cmu.edu

Yilei Qian, Brian Wichers, Qieyun Dai
Gui Ming Tang, Jerry Jiang
Twitter, Inc.

Jimmy Lin
University of Waterloo
jimmylin@uwaterloo.ca

ABSTRACT

Personalized recommendation products at Twitter target a multitude of heterogeneous items: Tweets, Events, Topics, Hashtags, and users. Each of these targets varies in their cardinality (which affects the scale of the problem) and their "shelf life" (which constrains the latency of generating the recommendations). Although Twitter has built a variety of recommendation systems before dating back a decade, solutions to the broader problem were mostly tackled piecemeal. In this paper, we present SimClusters, a general-purpose representation layer based on overlapping communities into which users as well as heterogeneous content can be captured as sparse, interpretable vectors to support a multitude of recommendation tasks. We propose a novel algorithm for community discovery based on Metropolis-Hastings sampling, which is both more accurate and significantly faster than off-the-shelf alternatives. SimClusters scales to networks with billions of users and has been effective across a variety of deployed applications at Twitter.

ACM Reference Format:
Venu Satuluri, Yao Wu, Xun Zheng, Yilei Qian, Brian Wichers, Qieyun Dai, Gui Ming Tang, Jerry Jiang, and Jimmy Lin. 2020. SimClusters: Community-Based Representations for Heterogeneous Recommendations at Twitter. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3394486.3403370

1 INTRODUCTION

Personalized recommendations lie at the heart of many different technology-enabled products, and Twitter is no exception. Our high-level goal is to make content discovery effortless and to free the user from the need for manual curation. On the Twitter platform, a wide variety of content types are displayed in a multitude of contexts, requiring a variety of personalization approaches. For example, recommendations of interesting Tweets are an essential component of not only the Home tab, but also for dissemination via email or push notifications. The "Who To Follow" module with user follow recommendations is crucial, especially for new users, and was one of the first recommendations products to be launched on Twitter [11]. Trends and Events (previously called Moments) are essential for informing the user about currently ongoing news stories and topics of conversation on the platform, and the Explore tab shows the user a personalized list of these. The recently launched Topics feature[1] lets users follow Topics (such as "Machine Learning" or "K-Pop") and see the algorithmically curated best tweets about those Topics in their Home feed.

Recommendation Problem (i.e., target)	Item Cardinality	Shelf Life	Display Location
User recommendations	$\sim 10^9$	Weeks	Who To Follow
Topic Tweet recommendations	$\sim 10^8$	Hours	Home, Explore
Out of network Tweet recommendations	$\sim 10^8$	Hours	Home
Similar Tweet recommendations	$\sim 10^8$	Hours	Tweet details page
Event recommendations	$\sim 10^3$	Hours	Explore
Trending Hashtags	$\sim 10^5$	Hours	Explore

Table 1: A partial list of recommendations problems at Twitter along with the number of possible recommendable items, the shelf life of the recommendations, and where they are shown on Twitter.

A summary of the diversity of personalized recommendation problems at Twitter is presented in Table 1. In all cases, we are making recommendations to users, but *what* we're recommending can be heterogeneous. For example, we could be suggesting users, Tweets, Events, Topics, or hashtags. There are two main dimensions of interest across the different recommendations problems: the cardinality of the items being recommended and the shelf life of the computed recommendations. The shelf life of our computed recommendations is closely related to the churn we observe in the recommended content. For example, since the follower graph changes relatively slowly, user follow recommendations can remain relevant for weeks (and thus can be computed in batch). At the other end of the spectrum, Tweet recommendations become stale much quicker and must be generated in an online system in close to real time; for this case, batch computations would not yield results fast enough to meet the product requirements. Naturally, in the case of recommendation problems where the item cardinality is large, being able to handle the scale of the problem is important.

*Work was done while interning at Twitter.

[1] https://blog.twitter.com/en_us/topics/product/2019/introducing-topics.html

KDD '20, August 23–27, 2020, Virtual Event, CA, USA

ACM ISBN 978-1-4503-7998-4/20/08…$15.00
https://doi.org/10.1145/3394486.3403370

Previously, Twitter built systems to tackle each of these different recommendation problems individually with little re-use or commonality. The original example here is the "Who To Follow" system that launched a decade ago [11] for user recommendations. Subsequently, Gupta et al. [12] described a specialized system to generate Tweet recommendations in real time, insights from which were later deployed in GraphJet [31]. GraphJet ingested the real-time stream of user-Tweet engagements to maintain a user-Tweet bipartite graph from which to generate recommendations, but the system was expensive to extend to new use cases. These aforementioned infrastructures were built mainly to generate candidates which got blended and scored subsequently. Twitter also built custom infrastructure for feature retrieval and scoring of arbitrarily generated candidates - examples include RealGraph [14] and RecService [9]. All of these systems were built with the aim of solving specific sub-problems in the recommendations landscape at Twitter and require separate development and maintenance. The central motivating question of this paper is: can we build a general system that helps us advance the accuracy of all or most of the Twitter products which require personalization and recommendations?

The solution proposed in this paper is built on the insight that we can construct, from the user–user graph, a general-purpose representation based on community structure, where each community is characterized by a set of influencers that many people in that community follow. Each of the different kinds of content (i.e., the targets in Table 1) is represented as a vector in the space of these communities, with the entry corresponding to the i-th community for item j indicating how interested the i-th community is in item j. The end result is that we can represent heterogeneous recommendation targets as sparse, interpretable vectors in the same space, which enables solutions for a wide variety of recommendation and personalization tasks (see details in Section 6). There are two notable aspects of our design:

(1) We avoid conventional matrix factorization methods that typically require solving massive numerical optimization problems, and instead rely on a combination of similarity search and community discovery, both of which are easier to scale. A key algorithmic innovation of our work is a new approach to community discovery — called Neighborhood-aware MH — which is 10×-100× faster, 3×-4× more accurate than off-the-shelf baselines, and scales easily to graphs with $\sim 10^9$ nodes and $\sim 10^{11}$ edges. It helps us discover $\sim 10^5$ communities on Twitter that are either organized around a common topic (e.g., "K-Pop" or "Machine Learning") or based on social relationships (e.g., those who work together or went to high-school together). We have open-sourced the implementation of the new algorithm in https://github.com/twitter/sbf.

(2) Our overall architecture has a modular and extensible design to enable the use of whichever computing paradigm is most suited to a specific component – batch-distributed, batch-multicore, or streaming-distributed. In particular, the ability to dynamically update representations using streaming-distributed components has proved crucial for accurately modeling Tweets which are Twitter's most important type of content.

We refer to our overall system as **SimClusters** (**Sim**ilarity-based **Clusters**) and have deployed it in production for more than a year. SimClusters also has the following features, which correspond to our design requirements:

(1) Universal representations: SimClusters provides representations for both users and a variety of content in the same space. This removes the need to invest in expensive custom infrastructure for each type of content.
(2) Computational scale: We are able to apply SimClusters at Twitter scale, with $\sim 10^9$ users, $\sim 10^{11}$ edges between them, and 10^8 new Tweets every day with $\sim 10^9$ user engagements per day.
(3) Accuracy beyond the head: SimClusters representations are accurate beyond just the most popular content ("head"), primarily due to the ability to scale SimClusters to a very large representational space with $\sim 10^5$ dimensions.
(4) Item and graph churn: The modular design of SimClusters makes it easy to extend to dynamic items which rapidly rise and diminish in popularity. Many of our important recommendations and engagement prediction problem involve items that churn rapidly – most Tweets, Events, and Trends stay relevant for no more than a day or two, meaning that it is crucial to be able to efficiently learn representations of new items before they lose their relevance.
(5) Interpretability: SimClusters representations are sparse and each dimension corresponds to a specific community, making them interpretable to a degree that is hard to obtain with alternatives such as matrix factorization or graph embeddings.
(6) Efficient nearest neighbor search: Identifying nearest neighbors is core to many downstream tasks such as generating recommendations, similar item retrieval, and user targeting. The sparsity of SimClusters representations makes it easy to setup and maintain inverted indices for retrieving nearest neighbors, even for rapidly churning domains (see details in Section 4).

SimClusters has been applied to many recommendations and personalization problems at Twitter — even for mature products such as out-of-network Tweet recommendations and Personalized Trends, SimClusters has enabled double digit improvements in the engagement rates of recommendations. It has also accelerated the building of entirely new products, such as Similar Tweets and Topic Tweet recommendations. SimClusters continues to be actively developed internally and applied to new use cases.

2 OVERVIEW OF SIMCLUSTERS

The SimClusters system (see Figure 1) consists of two stages:

(1) In the first stage (detailed in Section 3), we discover bipartite communities from our user-user graph at scale, resulting in learning sparse, non-negative representations for our users. At the end of this stage, each user is associated with a list of communities they participate in, along with the scores quantifying the strength of their affiliation to each of those communities. We refer to this output as "User Interest Representations" and it is made available in both offline data warehouses as well as low-latency online stores, indexed by the user id. This first stage is run in a batch-distributed setting, typically as a series of MapReduce jobs running on Hadoop.
(2) The second stage (detailed in Section 4) consists of several jobs running in parallel, each of which calculates the representations

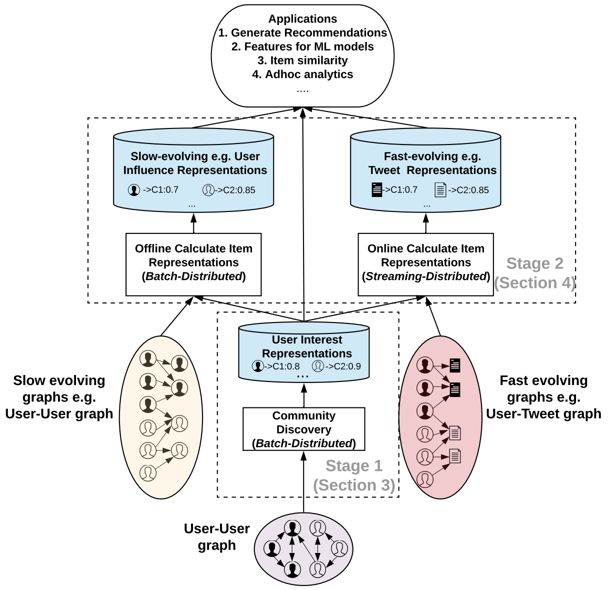
Applications
1. Generate Recommendations
2. Features for ML models
3. Item similarity
4. Adhoc analytics
....
Slow-evolving e.g. User Influence Representations
->C1:0.7
->C2:0.85
...
Fast-evolving e.g. Tweet Representations
->C1:0.7
->C2:0.85
...
Offline Calculate Item Representations (Batch-Distributed)
Online Calculate Item Representations (Streaming-Distributed)
Stage 2 (Section 4)
User Interest Representations
->C1:0.8
->C2:0.9
...
Community Discovery (Batch-Distributed)
Stage 1 (Section 3)
Slow evolving graphs e.g. User-User graph
Fast evolving graphs e.g. User-Tweet graph
User-User graph

Figure 1: Overview of SimClusters. Stage 1 detects bipartite communities from the user-user graph, which is fed to Stage 2 which runs several item representation jobs in parallel. For illustration, the color of the users and items in this figure indicates what community they get assigned to. In actuality, both users and items can participate in multiple communities.

for a specific recommendation target, using a user–target bipartite graph formed from interaction logs on the platform. Each job in the second stage operates in either a batch-distributed setting or a streaming-distributed setting, depending on the shelf-life of the recommendation target and the churn in the corresponding user–target bipartite graph.

The most important detail about our design is that it's based on discovering communities from the user–user graph. While the other user–target graphs on Twitter evolve rapidly, the user–user graph is relatively long-term and stable, and the specific communities discovered from the graph often outlive specific edges or nodes in the graph. In addition, the user-user graph usually also has more coverage, in the sense that there are a lot more users who have a minimum number of edges in this graph compared to the other user-item graphs.

The other important aspect of our design is its modularity. The different parts of the pipeline depend on each other only via offline data sets or online key-value stores, meaning that they are robust to delays in the preceding steps. It is also easy to swap out existing implementations with new variants, or run multiple implementations in parallel, as long as the output is in the format expected by the downstream jobs. The system degrades gracefully in the presence of bugs and errors - if any one of the item representation jobs has an issue, the other item representation jobs can still service their applications. Our design also allows for gradually adding more modules to the system without needing to build it all in at the start — in fact, the first version of the system only output communities without any item representations, but this by itself had many useful applications.

3 STAGE 1: COMMUNITY DISCOVERY

This stage is about discovering communities from the Twitter user–user graph i.e. the directed graph of Follow relationships between users. Following seminal work in the analysis of directed graphs such as HITS [17] and SALSA [20], we find it convenient to reformulate the directed graph as a bipartite graph. We now frame our task as one of identifying bipartite communities i.e. communities consisting of members from left as well as right partitions, and where the edge density between the left and right member sets is high. The bipartite reformulation lets us more flexibly assign users to communities — similar to HITS, we decouple the communities a user is influential in from the communities in which a user is interested.

Problem Definition 1. *Given a bipartite user–user graph with left-partition L and right-partition R, find k (possibly overlapping) bipartite communities from the graph, and assign each left-node and right-node to the communities with weights to indicate the strength of their memberships.*

The other advantage of reformulating the directed graph as a bipartite graph is that we can choose to make R, the right set of nodes, different from L, the left set of nodes – in particular, since the majority of edges in a typical social network is directed towards a minority of users, it makes sense to pick a smaller R than L. In Twitter's case, we find that we're able to cover the majority of edges (numbering $\sim 10^{11}$) in the full graph by including the top $\sim 10^7$ most followed users in R, while L continues to include all users, which is $\sim 10^9$. Our problem definition also asks to assign non-negative scores to both the left and the right members indicating the strength of association to a community. Therefore, we represent the left and right memberships as sparse, non-negative matrices $\mathbf{U}_{|L| \times k}$ and $\mathbf{V}_{|R| \times k}$, where k is the number of communities. Hence, the problem of bipartite community discovery bears close similarities to the problem of sparse non-negative matrix factorization (NMF). The biggest challenge with adapting existing approaches such as NMF and their variants [1, 3, 35] is the inability to scale them to graphs with $\sim 10^9$ nodes and $\sim 10^{11}$ edges – all of our attempts internally to adapt these approaches (see e.g. [30]) have only worked at smaller scales and have been very difficult to debug and maintain.

With SimClusters, we instead adopt the following 3-step approach, illustrated with a toy example in Figure 2.

(1) **Similarity Graph of Right Nodes:** We calculate the "right projection" of the bipartite graph, i.e., we calculate the similarity between the nodes in the right partition of the bipartite graph based on their incoming edges, and we form a weighted, undirected graph consisting only of the nodes in the right partition. More details in Section 3.1.
(2) **Communities of Right Nodes:** We discover communities from this similarity graph, using a *novel* neighborhood-based sampling algorithm that is inspired by the work of [33] but is much more accurate, faster, and scales to graphs with billions of edges. More details in Section 3.2.

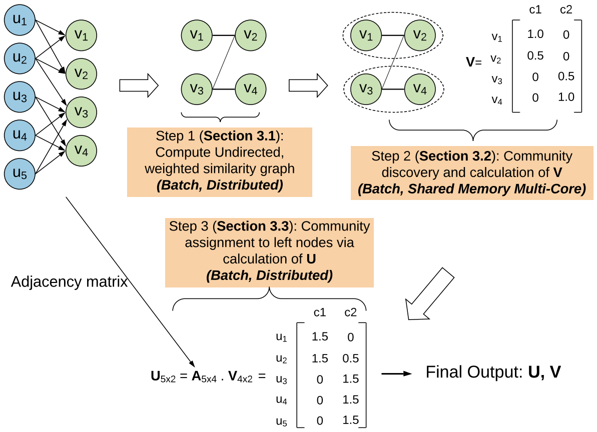
u1
u2
u3
u4
u5
v1
v2
v3
v4
Step 1 (Section 3.1): Compute Undirected, weighted similarity graph (Batch, Distributed)
Step 2 (Section 3.2): Community discovery and calculation of V (Batch, Shared Memory Multi-Core)
V=
c1 c2
v1 1.0 0
v2 0.5 0
v3 0 0.5
v4 0 1.0
Adjacency matrix
Step 3 (Section 3.3): Community assignment to left nodes via calculation of U (Batch, Distributed)
U5x2 = A5x4 . V4x2 =
c1 c2
u1 1.5 0
u2 1.5 0.5
u3 0 1.5
u4 0 1.5
u5 0 1.5
Final Output: U, V

Figure 2: Overview of 3-step approach to discovering bipartite communities (Stage 1).

(3) **Communities of Left Nodes:** We now assign nodes from the left partition to the communities discovered in step 2, and this is described in Section 3.3.

We note that the broad outlines of this approach have been independently discovered and suggested in the past literature (see e.g. [23, 28]), but it has been largely neglected as an option among similar deployments in industry. The primary reason we think this works is that while the original bipartite graph is massive and noisy, the similarity graph of right nodes is much smaller and has clearer community structure. From a scalability point of view, this approach shifts most of the computational burden to the first step of identifying pairs of similar users based on their followers, which is a problem that is largely solved at Twitter (see Section 3.1). From a matrix-factorization point of view, this 3-step approach closely mirrors one way of performing SVD of a matrix $\mathbf{A}$, via the eigen-decomposition of $\mathbf{A}^T\mathbf{A}$.

Our 3-step approach also isolates the hard-to-parallelize step of community discovery into step 2, where it operates on a smaller graph that fits into the memory of a single machine, while the other two steps operate on much bigger inputs and out of necessity run in batch-distributed settings such as Hadoop MapReduce. This approach is also modular and allows for swapping out implementations of each of the above steps independent of the others. A possible concern with our 3-step approach is that it may lead to reduced accuracy compared to directly learning the communities on the input bipartite network – we empirically test this in Supplemental Section A.2 and find this not to be an issue.

3.1 Step 1: Similarity Graph of Right Nodes

The goal of this step is to construct a much smaller unipartite, undirected graph G over the nodes of the right partition. We define the weight between two users (u, v) based on the *cosine similarity* of their followers on the left side of the bipartite graph. To elaborate, if $\vec{x_u}$ and $\vec{x_v}$ represent the binary incidence vectors of u's and v's followers, their cosine similarity is defined as $\vec{x_u} \cdot \vec{x_v} / \sqrt{\|\vec{x_u}\| \|\vec{x_v}\|}$. With this definition, two users would have non-zero similarity, or an edge in G simply by sharing one common neighbor in the bipartite graph. In order to avoid generating an extremely dense similarity graph, we discard the edges with similarity score lower than a certain threshold and additionally keep at most a certain number of neighbors with the largest similarity scores for each user.

The difficulty is that solving the similar users problem is very challenging at Twitter scale. But because this is a problem with important applications – e.g. it is the foundation of applying item-based collaborative filtering for the "Who To Follow" module [11] – we have invested significant resources to develop a robust solution. Our solution, called `WHIMP`, uses a combination of wedge sampling and Locality Sensitive Hashing (LSH) to scale to the Twitter graph and lends itself to implementation on Hadoop MapReduce [32]. `WHIMP` is able to identify similar users for users with either large or small followings, and has been vetted in a variety of ways internally.

Ultimately, this similarity graph step takes as input a directed/bipartite graph with $\sim 10^9$ nodes and $\sim 10^{11}$ edges and outputs an undirected graph with $\sim 10^7$ nodes and $\sim 10^9$ edges. In other words, we go from shared-nothing cluster-computing scale to shared-memory multi-core scale. The transformation wrought by this step is also reminiscent of prior research which suggested that keeping only the most important edges in a graph can benefit community discovery methods [29].

3.2 Step 2: Communities of Right Nodes

In this step, we wish to discover communities of densely connected nodes from the undirected, possibly-weighted similarity graph from the previous step. In order to accurately preserve the structure of the input similarity graph, we have observed that it is important for the communities to have hundreds of nodes, rather than thousands or tens or thousands. This means that we need algorithms that can process input graphs with $\sim 10^7$ nodes and $\sim 10^9$ edges to find $\sim 10^5$ communities. Despite the long history of community discovery algorithms, we were unable to find any existing solution that can satisfy these scale requirements. We next describe the algorithm we developed, called Neighborhood-aware Metropolis Hastings (henceforth Neighborhood-aware MH), to meet our requirements.

Our algorithm extends a Metropolis-Hastings sampling approach presented in [33] for discovering overlapping communities, which we first describe as background. Let $\mathbf{Z}_{|R| \times k}$ be a sparse binary community assignments matrix and $\mathbf{Z}(u)$ denote the set of communities to which the vertex u has been assigned (in other words, $\mathbf{Z}(u)$ gives the non-zero column indices from the u-th row in $\mathbf{Z}$). Equation 1 specifies an objective function over $\mathbf{Z}$.

$$\mathcal{F}(\mathbf{Z}) \overset{\text{def}}{=} \alpha \sum_{(u,v) \in E} \mathbb{1}(|\mathbf{Z}(u) \cap \mathbf{Z}(v)| > 0) + \sum_{(u,v) \notin E} \mathbb{1}(|\mathbf{Z}(u) \cap \mathbf{Z}(v)| = 0) \tag{1}$$

$\mathbb{1}$ is the indicator function. $\mathcal{F}(\mathbf{Z})$ is the sum of two terms – the first counts how many neighboring pairs of nodes in the graph share at least one community, while the second counts how many non-neighbor pairs of nodes in the graph do not share a community.[2] Since most real, large-scale networks are very sparse, it is useful to upweight the contribution of the first term using the parameter α – increasing values of α means that the objective function is

[2] We give details here assuming the input network is unweighted, to simplify the exposition, but the method is easy to extend (conceptually at least) to the case where the input graph has weights.

Algorithm 1: Generalization of the Metropolis-Hastings approach to community discovery from [33]

```
 1: Input: Graph G, desired communities $k$, hyper-parameter $T$
 2:   $\mathbf{Z} \leftarrow$ Initialize($G, k$)
 3:   for $t \leftarrow 1$ to $T$ epochs do
 4:     for $u \leftarrow$ randomShuffle($1 \ldots |V|$) do
 5:       $\mathbf{Z}'(u) \leftarrow$ Proposal($u, G, \mathbf{Z}, k$)
 6:       $p \leftarrow min(1, e^{f(u,\mathbf{Z}') - f(u,\mathbf{Z})})$ // $p$ is 1 iff
          $f(u, \mathbf{Z}') > f(u, \mathbf{Z})$
 7:       With probability $p$, accept $\mathbf{Z}'(u)$ i.e. set $\mathbf{Z}(u) \leftarrow \mathbf{Z}'(u)$
 8:     end for
 9:   end for
10: return $\mathbf{Z}$
```

Algorithm 2: Initialize and Proposal functions for Random MH [33]

```
1: Function: Initialize($G, k$)
2: return $|V| \times k$ binary matrix where each row is a uniformly
   random sample from the $2^k$ space of length-$k$ binary vectors
3:
4: Function: Proposal($u, G, \mathbf{Z}, k$)
5: return uniformly random sample from $2^k$ space of length-$k$
   binary vectors
```

Algorithm 3: Initialize and Proposal functions for Neighborhood-aware MH

```
 1: Function: Initialize($G, k$)
 2:   for $i \leftarrow 1..k$ do
 3:     Set $i^{th}$ column of $\mathbf{Z}$ as neighbors of a randomly picked
        node
 4:   end for
 5: return $\mathbf{Z}$
 6:
 7: Function: Proposal($u, G, \mathbf{Z}, k, l$) // $l << k$
 8:   $S \leftarrow$ columns of $\mathbf{Z}$ with $\geq 1$ non-zero in rows of $N(u)$
      // enumerateSubsets($S, l$) returns all subsets of $S$ of size $\leq l$
 9:   for $s \leftarrow$ enumerateSubsets($S, l$) do
10:     fMap($s$) $\leftarrow f(u, s)$ // Per Eqn 2
11:   end for
12: return Sample $s$ from $S$ according to softmax(fMap)
```

better optimized by $\mathbf{Z}$ with more non-zeros. Note also that the objective function above is decomposable, in the sense that the overall objective function $\mathcal{F}(\mathbf{Z})$ can be expressed as a sum of a function $f(u, \mathbf{Z})$ over individual vertices (below, $\mathcal{N}(u)$ denotes the set of neighbors of vertex u).

$$f(u,\mathbf{Z}) \overset{\text{def}}{=} \alpha \sum_{v \in \mathcal{N}(u)} \mathbb{1}(|\mathbf{Z}(u) \cap \mathbf{Z}(v)| > 0) + \sum_{v \notin \mathcal{N}(u)} \mathbb{1}(|\mathbf{Z}(u) \cap \mathbf{Z}(v)| = 0) \tag{2}$$

Using the above background, we first describe the approach for discovering overlapping communities in a general way in Algorithm 1. After initializing $\mathbf{Z}$, we run at most T epochs of optimization, where in each epoch we iterate over all the vertices in the graph in a shuffled order. For each vertex u we sample a new set of community assignments $\mathbf{Z}'(u)$ using the proposal function, and calculate the difference in objective function between the newly proposed $\mathbf{Z}'(u)$ and the current set of community assignments $\mathbf{Z}(u)$. If $\mathbf{Z}'(u)$ is better, then it is accepted; if not, it may still be accepted with a certain probability, indicated in line 6 of Algorithm 1. As noted in [33], one reason for preferring a randomized optimization procedure as opposed a deterministic optimization procedure is to avoid getting stuck in local minima.

The specific choices for the 'Initialize' and 'Proposal' functions made in [33] are described in Algorithm 2. Because these functions are implemented using purely random sampling, we refer to this approach as 'Random MH'. The main practical drawback of Random MH is that it is extremely slow to obtain a satisfactorily accurate solution for even moderate values of k. This is not surprising considering that in each step, the proposal function generates a completely random community assignments vector and evaluates it w.r.t. the current vector; as k increases, the space of community assignments increases exponentially which makes it very unlikely that the proposal will be able to generate an acceptable transition.

Instead, we propose Neighborhood-aware MH, specified in Algorithm 3. The proposal function in Neighborhood-aware MH is based on two insights or assumptions – the first is that it is extremely unlikely that a node should belong to a community that none of its neighbors currently belongs to; the second is that for most practical applications, it is unnecessary to assign a node to more than a small number of communities. We design a two-step proposal function that works as follows. In the first step, for a given node u we iterate over all the neighbors of u, look up their community assignments in $\mathbf{Z}$, and identify the set of communities which are represented at least once, call it S. In the second step, we iterate over all subsets of size $\leq l$ of S from the first step, where l is a user-provided upper bound on how many communities a node can be assigned to. For each subset s, we calculate the function $f(u, s)$ from Eqn 2, and finally sample the subset s with probability proportional to $e^{f(u,s)}$ i.e. we apply the softmax. The result of the sampling is then either accepted or rejected, as specified in lines 6 and 7 of Algorithm 1. As for initializing $\mathbf{Z}$, we seed each community with the neighborhood for a randomly selected node in the graph.

We discuss a few important implementation details.

- Most of the complexity comes from evaluating the function $f(u, s)$, which requires calculating the intersection between a node's neighbors and the union of the communities in s. For many members of S (the set computed in line 8 Algorithm 3), we can incrementally compute the summary statistics required for $f(u, s)$ as we go through a node's neighborhood when executing line 8 of Algorithm 3, so that the subsequent inner loop in line 10 can execute much faster. Similarly, the acceptance probability for line 6 of Algorithm 1 can also reuse the $f(u, \mathbf{Z})$ computed during the proposal process.
- Sampling from a softmax distribution can be accomplished efficiently in a single pass using the Gumbel-Max trick.[3]

[3] http://timvieira.github.io/blog/post/2014/07/31/gumbel-max-trick/

- In the important special case where we assign each node to at most one community only, each epoch of Neighborhood-aware MH can execute in $O(|E|)$ time, using both of the above mentioned tricks.
- The algorithm lends itself well to parallelization. Specifically the for loop in line 4 of Algorithm 1 can be distributed among several threads which share access to $\mathbf{Z}$, the rows of which can optionally be synchronized using read-write locks. In practice, we have found that removing synchronization has no effect on the accuracy and gives a slight boost in speed (similar to [24]).

3.3 Step 3: Communities of Left Nodes

The output of the previous step is the matrix $\mathbf{V}_{|R| \times k}$ in which the i-th row specifies the communities to which the right-node i has been assigned. The remaining problem that needs to be solved is coming up with the matrix $\mathbf{U}_{|L| \times k}$ such that the i-th row specifies the communities to which the left-node i has been assigned. A simple way to do this assignment is to assign a left-node to communities by looking at the communities that its neighbors (which will all be right-nodes, and hence already have assignments) have been assigned to. More formally, if $\mathbf{A}_{|L| \times |R|}$ is the adjacency matrix of the input bipartite graph, then we set $\mathbf{U} = truncate(\mathbf{A} \cdot \mathbf{V})$, where the *truncate* function keeps only up to a certain number of non-zeros per row to save on storage. This equation for calculating $\mathbf{U}$ is motivated by the fact that in the special case when $\mathbf{V}$ is an orthonormal matrix, i.e. $\mathbf{V}^T\mathbf{V} = I$, then $\mathbf{U} = \mathbf{A} \cdot \mathbf{V}$ is the solution to $\mathbf{A} = \mathbf{U} \cdot \mathbf{V}^T$. We have experimented with situations both where $\mathbf{V}$ is orthonormal (this can be achieved by assigning each right-node to at most one community) as well as situations where $\mathbf{V}$ is not, and have found that in each case the resulting $\mathbf{U}$ provides accurate representations for the left-nodes. We refer to $\mathbf{U}$ as User Interest Representations, and it forms the main input for subsequent steps. The computation in this step can be scaled to our requirements easily by implementing in a batch-distributed computing paradigm such as Hadoop MapReduce.

4 STAGE 2: ITEM REPRESENTATIONS

In this section, we describe how to compute representations for different items, such as Tweets, Hashtags, or users - which can be the targets for different recommendation problems. Along with the user interest representations $\mathbf{U}$ from Stage 1, this stage also relies on a user–item bipartite graph that is formed from historical or on-going user engagements with those items on the platform.

Our general framework is to compute an item's representation by aggregating the representations of all the users who engaged with it, i.e., the representation for item j is

$$\mathbf{W}(j) = aggregate(\{\mathbf{U}(u), \forall u \in \mathcal{N}(j)\}), \tag{3}$$

where $\mathcal{N}(j)$ denotes all the users who engaged with item j in the corresponding user–item bipartite graph, and $\mathbf{W}(j)$ and $\mathbf{U}(u)$ are both vectors. The *aggregate* function can be chosen based on different applications, and can also be learned from a specific supervised task [13]. In our case, we opt for a relatively simple, interpretable *aggregate* function with the goal that $\mathbf{W}(j, c)$ can be interpreted as the level of interest an average user of the community c currently has in this item j. We choose to use “exponentially time-decayed average” as our *aggregate* function, which exponentially decays the contribution of a user who interacted with the item based on how long ago that user engaged with the item. The half-life used for the exponential decay is item-dependent – where the shelf-life of those items is longer (such as Topics), we set longer half-lives, while for shorter shelf life items such as Tweets, we set shorter half-lives.

Algorithm 4: Updating item representations in streaming setting

```
1:  Input:
      • Parameter $half\_life$ for the exponential decay,
      • edge $(u, j, t)$ indicating user $u$ engaged with item $j$ at time $t$,
      • User Interest representations $\mathbf{U}$
2:  Input and Output:
      • Distributed cache $\mathbf{W}$ where keys are pairs of (item,
        community) and values are (last_value, last_time),
      • Distributed cache $\mathcal{R}$ where keys are items and values are
        top communities and their scores (with last_time),
      • Distributed cache $\mathcal{C}$ where keys are communities and
        values are top items and their scores (with last_time).
3:  for $c \leftarrow$ community ids of $\mathbf{U}(u)$ with nonzero values do
4:    $(last\_value, last\_time) \leftarrow \mathbf{W}(j, c)$
5:    $decay\_factor \leftarrow 2^{-(t - last\_time)/half\_life}$
6:    $new\_value \leftarrow decay\_factor \times last\_value + 1$
7:    $\mathbf{W}(j, c) \leftarrow (new\_value, t)$
8:    $\mathcal{R}(j) \leftarrow updateTopK(\mathcal{R}(j), \mathbf{W}(j, c))$ // update the top
      communities and their scores (and last_time) based on the
      updated $\mathbf{W}(j, c)$
9:    $\mathcal{C}(c) \leftarrow updateTopK(\mathcal{C}(c), \mathbf{W}(j, c))$ // update the top items
      and their scores (and last_time) based on the updated $\mathbf{W}(j, c)$
10: end for
```

The resulting matrix $\mathbf{W}$ is much denser than $\mathbf{U}$ and it is not useful to save all its non-zero values at scale. Instead, we maintain two additional views or indexes of $\mathbf{W}$, each of which keeps a TOP-K view. The first view is $\mathcal{R}$ and $\mathcal{R}(j)$ tracks the top communities for the item j. The second view is $\mathcal{C}$ and $\mathcal{C}(c)$ tracks the top items for the community c. In the case of items with a long shelf life, the calculation of $\mathbf{W}$, $\mathcal{R}$, and $\mathcal{C}$ is straightforwardly done in a batch setting using e.g. Hadoop MapReduce.

However, handling items with short shelf life is more interesting. In this case, we realize a major advantage of an exponentially time-decayed average (as opposed to e.g. time-windowed average), which is that it lends itself to easy incremental updates for $\mathbf{W}$. Specifically, we just need to keep two summary statistics for each cell in $\mathbf{W}$ - the current average itself and the last timestamp when it was updated. As detailed in Algorithm 4 lines 4–7, when a new user–item engagement arrives, we are able to update $\mathbf{W}$ for the item by calculating a decay factor based on the time elapsed since the last update. In order to exactly track the row-wise and column-wise top-k views on $\mathbf{W}$, it is necessary that we track the entirety of $\mathbf{W}$ - if it turns out that $\mathbf{W}$ is too big to be tracked in its entirety, then one can use sketches to keep a summary of $\mathbf{W}$ at the cost of introducing errors [4, 15], although we have found this unnecessary. Another way of reducing the size of $\mathbf{W}$ is to reduce k i.e. the dimensionality of the representations computed in Stage 1, or by further sparsifying the input user representations $\mathbf{U}$. Calculating streaming item representations in this manner can be implemented using frameworks such as Apache Storm/Heron/Spark/Flink.

The two top-k views $\mathcal{R}$ and C are stored in low-latency key-value stores. Using these two indices, it easy to retrieve nearest neighbors for any user or item – we simply look up the top communities that a user or item is active in, and for each of those communities, identify the top users or items. These candidates can then be ranked by fetching their full representations and computing the similarity with the representation of the query object (either user or item). The upshot is that we neither need to brute-force scan through all users/items nor need to build specialized nearest neighbor indices.

5 DEPLOYMENT DETAILS

SimClusters has been deployed in production at Twitter for more than an year so far. All the representations output by the SimClusters system are also keyed by *model-version*, so that we can operate multiple models in parallel to enable the trying out of new parameters or code changes without affecting existing production. The main model that is currently running in production has $\sim 10^5$ communities in the representations, discovered from the similarity graph of the top $\sim 10^7$ users by follower count. The bipartite communities discovered by the model contain nearly 70% of the edges in the input bipartite graph, suggesting that most of the structure of the graph is captured. The right member sets do not vary too much in their sizes, while the left member sets vary drastically, reflecting the variance in the original follower distribution. Within Stage 1, Step 1 (similarity calculation) is the most expensive step, taking about 2 days to run end-to-end on Hadoop MapReduce, but note that this job was in production before SimClusters and therefore is not an additional cost introduced by SimClusters. Step 2 was run from scratch for the very first time when SimClusters launched; subsequently, we update the $\mathbf{V}$ matrix to take into account changes to the user–user similarity graph by running an abbreviated version of Neighborhood-aware MH initialized with the current $\mathbf{V}$. Step 3 is also periodically run as batch application on Hadoop MapReduce using the latest version of the user–user graph and the latest $\mathbf{V}$ from Step 2. Once we have the output from Step 3 (the $\mathbf{U}$ matrix), we do not directly use the $\mathbf{V}$ matrix anymore, which is typically too sparse for accurate modeling.

For Stage 2, we currently have four jobs – two batch jobs, one for "user influence" representations and one for Topic representations; and two streaming jobs, one for Tweet representations and one for Trend representations. The purpose of the user influence representations is to tell us what communities a user is influential in, as opposed to the user interest representations (the output of Stage 1) which tell us what communities a user is interested in, The user influence representations are better than the original original $\mathbf{V}$ matrix for this purpose as they cover many more users and are also denser for the original subset of users. Topic representations tell us which communities are the most interested in a Topic, and the input to computing these is both the user interest representations as well as a user–Topic engagement graph. Tweet and Trend representations are computed and updated in a streaming job which takes as input user–Tweet engagements happening in real-time. Both the user interest and user influence representations are protected using authentication to only allow authorized access, and users are provided the chance to opt out of unwanted inferences in their Privacy dashboard.

Note that we store only the non-zeros in all our representations, and in all cases we truncate entries close to zero. The user interest representations cover $\sim 10^9$ users while the user influence representations cover $\sim 10^8$ users, with both representations having on average $10 - 100$ non-zeros. There are fewer recommendable Tweets and Trends at any given point in time (refer Table 1), but their representations are denser, having on average $\sim 10^2$ non-zeros. Note that for the following four representations - user influence, Topic, Tweet, and Trend - we also maintain the inverted indices, i.e. given a community, what are the top-k users/Topics/Tweets/Trends for that community (denoted by C in Section 4). Having C is essential to retrieving the items whose representation has the largest dot product or cosine similarity with another representation.

6 APPLICATIONS

6.1 Similar Tweets on Tweet Details page

For users who visit a Tweet via an email or a push notification, Twitter shows a module with other recommended Tweets, alongside replies. Prior to SimClusters, this module retrieved Tweets solely based on author similarity i.e. Tweets written by users who share a lot of followers with the author of the main Tweet on the page. We ran an online A/B test where we added similar Tweets from SimClusters i.e. we retrieved Tweets whose SimClusters representation has high cosine similarity with the representation of the main Tweet on the page. We found that the engagement rate on the resulting Tweets was 25% higher.[4]

Subsequently, we added a second candidate source for this product based on SimClusters – retrieve Tweets whose SimClusters representation have high cosine similarity with the user influence representation of the author of the main Tweet on the page. Adding this source increases the coverage further, while the overall increase in engagement rate is a more modest but still impressive 7%.

6.2 Tweet Recommendations in Home Page

A user's Home feed on Twitter consists of both Tweets from users being directly followed as well as recommended Tweets from users not being followed ("Out of Network Tweets"). Prior to SimClusters, the main algorithm for recommended Tweets was what is called "Network Activity" - namely, use GraphJet [31] to identify which Tweets are being liked by the viewing user's followings (i.e. network).

Using SimClusters Tweet representations, we built two candidate sources to supplement Network Activity Tweets. The first candidate source identifies Tweets whose real-time representation has the highest dot-product with the viewing user's interest representation. The second candidate source is based on item-based collaborative filtering, and uses the same underlying implementation as the "Similar Tweets" application described in Section 6.1 to identify Tweets similar to those Tweets which have been recently

[4] All the reported percentage changes in this section are statistically significant per our internal A/B testing tool.

liked by the user. We ran an online A/B test by replacing existing candidates in production (in certain positions on Home) using the candidates from these two new candidate sources. The experiment showed that the engagement rate of the new candidates is 33% higher than that for candidates generated by Network Activity, and shown in similar positions. The two candidate sources together were able to increase total weighted engagements on the platform by close to 1%, which is very large considering the maturity of this product and that recommended Tweets only account for a minority of the viewed content in Home pages.

Apart from new candidates, we also use the user interest and Tweet representations to improve the ranking of candidates coming from all sources. The user and item representations are used to enrich the set of existing user features, item features, as well as user-item interaction features in the input to an engagement prediction model. A/B testing showed that the model trained with these features was able to increase engagement rate of recommended content by 4.7% relatively, which is a significant lift for a mature model.

6.3 Ranking of Personalized Trends

Showing top trending content (e.g., Hashtags, Events, breaking news) is an important way to keep users informed about what is happening locally and globally. The implementation for Trends follows a two-stage process of Trends detection followed by ranking. Prior to SimClusters, the ranking of a Trend primarily depended on its volume and a small number of personalization features. We used Trends SimClusters representations to score Trends for a given user by using the dot-product of the user's interest representation along with the real-time representation for a Trend. A/B testing revealed that using these scores led to a 8% increase in user engagement with the Trends themselves, as well as a bigger 12% increase in engagement on the landing page subsequent to a click. These improvements are large when compared against other experiments run on this product.

6.4 Topic Tweet Recommendations

Given a Topic in a pre-defined topic taxonomy such as "Fashion" or "Marvel Movies", how can we identify the best content about it? The original implementation here (before the product was launched publicly) primarily relied on custom text matching rules curated by human experts to identify topical Tweets. Once we realized that this approach surfaced a number of false positives (primarily due to a Tweet's text incidentally matching the rules for a Topic), we tested a second implementation where we first identify those Tweets whose SimClusters representation has high cosine similarity with the representation of the query Topic, and then apply the textual matching rules. Internal evaluation showed that the second approach returned much better results, therefore we launched this product publicly using this approach. Since launch, this feature has received positive press externally as well as causing higher engagement with Tweets from the broader user base.

6.5 Ranking Who To Follow Recommendations

The candidates for Who To Follow recommendations are ranked using an engagement prediction model, to which we added new features based on the SimClusters representations of the viewing user and the candidate user. In A/B tests, we observed an impressive increase of 7% in the follow rate by using these new features.

6.6 Applications in progress

6.6.1 ***Notifications quality filter.*** A crucial task on Twitter is to protect users from getting abusive or spammy replies or mentions. We developed new SimClusters representations for users based on the user–user block graph (i.e. when one user blocks another), and used these representations as features to train a model for filtering out abusive and spammy replies. In offline tests, the model showed an impressive 4% lift in PR-AUC[5].

6.6.2 ***Supervised embeddings from feature combinations.*** While SimClusters representations mostly capture information from various engagement graphs, we are also experimenting approaches to combine it with other features about users or items (for example, follower counts or geo information). One approach where we are obtaining promising early results is to train a deep neural network on an ancillary prediction task (such as engagement prediction) where the input features are both the user and item SimClusters representations along with previously developed features for the user and item. By choosing the right architecture for this neural net, for example, the two-tower DNN model [36], we are able to learn dense embeddings separately for users and items.

6.6.3 ***Real-time Event notifications.*** A major application at Twitter is to notify users who may be interested when a major news event happens. Using the SimClusters representation of an Event (which is in turn derived by aggregating the representations of the human-curated Tweets about it), we can identify the communities of users who will be interested in it, and subsequently target users interested in them. We are currently evaluating such an approach.

7 RELATED WORK

Traditionally, approaches to recommender systems are categorized as either neighborhood-based (which do not involve model-fitting), or model-based (which fit a model to the input data).

In our experience of building recommendations at Twitter, we find that neighborhood-based methods are easier to scale, more accurate, more interpretable, and also more flexible in terms of accommodating new users and/or items [9, 11, 12, 31]. Recent research has also found that well-tuned neighborhood-based methods are not easy to beat in terms of accuracy [6]. However, neighborhood-based approaches do not provide a general solution – we needed to build and maintain separate systems to solve each recommendation sub-problems at Twitter in the past (see Section 1 for more discussion of our past work).

Model-based approaches, such as factorized models [18], graph embedding [10, 26] or VAE [22], fit separate parameters for each user or item. The number of model parameters that need to be learned in order to scale to a billion-user social network can easily approach 10^{12}, necessitating unprecedentedly large systems for solving ML problems at that scale. Hybrid models, such as Factorization Machine [27] and Deep Neural Networks (DNNs) [5] have been introduced to reduce the parameter space by utilizing the side information as prior knowledge for users and items. However, they

[5] Area Under Curve for Precision-Recall, a metric for classifiers

require either well-defined feature vectors or pre-trained embeddings from auxiliary tasks as the input representation of users and items. Graph Convolutional Networks (GCNs) [16, 37] can enrich pre-existing feature representations of the nodes by propagating the neighborhood information from the graph, without fitting model parameters for each node. GCNs perform well in domains where the items have a good set of pre-existing features, e.g., where the items are images [37]. Such approaches work less well in the absence of useful content features and cannot deal with the short half life of items either. We see SimClusters as an approach to scalably learn user and item representations which can be fed to hybrid models like DNNs [5] or GCNs [37].

Our problem definition bears some similarity to the cross-domain or heterogeneous recommender systems problem [2, 38], where one can use a joint objective function to simultaneously learn the representations of users and items across multiple domains [8, 39]. It is unclear how these methods can support our requirements for scale, handling dynamic items and graphs, and intepretability.

8 CONCLUSION

We proposed a framework called SimClusters based on detecting bipartite communities from the user-user graph and use them as a representation space to solve many personalization and recommendation problems at scale. SimClusters uses a novel algorithm called Neighborhood-aware MH for solving the crucial problem of unipartite community detection with better scalability and accuracy. We also presented several diverse deployed and in-progress applications where we use SimClusters representations to improve relevance at Twitter.

ACKNOWLEDGMENTS

The authors acknowledge Mary Linnell, Volodymyr Zhabiuk, Jiasong Sun, Rohit Jain, Wenzhe Shi, Arvind Thiagarajan, Michalis Rossides, Shaodan Zhai, Busheng Lou, Lohith Vuddemarri, Yang Tang, and Jun Xie for deploying SimClusters to the products at Twitter, as well as Suvash Sedhain, Yi-Chin Wu and Hafeezul Rahman Mohammad for providing valuable feedback and continuous contributions to this project.

REFERENCES

[1] Edoardo M. Airoldi, David M. Blei, Stephen E. Fienberg, and Eric P. Xing. 2008. Mixed Membership Stochastic Blockmodels. *JMLR* 9 (June 2008), 1981–2014.

[2] Iván Cantador and Paolo Cremonesi. 2014. Tutorial on Cross-domain Recommender Systems. In *RecSys '14*. 401–402.

[3] Andrzej Cichocki and Anh-Huy Phan. 2009. Fast Local Algorithms for Large Scale Nonnegative Matrix and Tensor Factorizations. *IEICE Transactions* 92-A (03 2009), 708–721.

[4] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[5] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *RecSys '16*. 191–198.

[6] Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. 2019. Are We Really Making Much Progress? A Worrying Analysis of Recent Neural Recommendation Approaches. In *Recsys'19*. 101–109.

[7] Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis. 2007. Weighted Graph Cuts Without Eigenvectors A Multilevel Approach. *IEEE Trans. Pattern Anal. Mach. Intell.* 29, 11 (Nov. 2007), 1944–1957.

[8] Ali Mamdouh Elkahky, Yang Song, and Xiaodong He. 2015. A multi-view deep learning approach for cross domain user modeling in recommendation systems. In *WWW'15*. 278–288.

[9] Ajeet Grewal, Jerry Jiang, Gary Lam, Tristan Jung, Lohith Vuddemarri, Quannan Li, Aaditya Landge, and Jimmy Lin. 2018. Recservice: Distributed Real-Time Graph Processing at Twitter. In *HotCloud'18*. USENIX Association, 3.

[10] Aditya Grover and Jure Leskovec. 2016. Node2Vec: Scalable Feature Learning for Networks. In *KDD '16*. 855–864.

[11] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. 2013. WTF: The Who to Follow Service at Twitter. In *WWW '13*. 505–514.

[12] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *Proceedings of the VLDB Endowment* 7, 13 (2014), 1379–1380.

[13] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NIPS'17*. 1025–1035.

[14] Krishna Kamath, Aneesh Sharma, Dong Wang, and Zhijun Yin. 2014. Realgraph: User interaction prediction at twitter. In *User Engagement Optimization Workshop at KDD'14*.

[15] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. 2003. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems (TODS)* 28, 1 (2003), 51–55.

[16] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *ICLR'17*.

[17] Jon M. Kleinberg. 1999. Authoritative Sources in a Hyperlinked Environment. *J. ACM* 46, 5 (Sept. 1999), 604–632.

[18] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (Aug. 2009), 30–37.

[19] Jérôme Kunegis. 2013. KONECT – The Koblenz Network Collection. In *Proc. Int. Conf. on World Wide Web Companion*. 1343–1350.

[20] R. Lempel and S. Moran. 2001. SALSA: The Stochastic Approach for Link-Structure Analysis. *ACM Trans. Inf. Syst.* 19, 2 (April 2001), 131–160.

[21] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.

[22] Dawen Liang, Rahul G. Krishnan, Matthew D. Hoffman, and Tony Jebara. 2018. Variational Autoencoders for Collaborative Filtering. In *WWW '18*. 689–698.

[23] David Melamed. 2014. Community Structures in Bipartite Networks: A Dual-Projection Approach. *PLOS ONE* 9, 5 (05 2014), 1–5.

[24] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. 2011. HOGWILD!: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS'11*. 693–701.

[25] F. et. al. Pedregosa. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[26] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *KDD'14*. 701–710.

[27] Steffen Rendle. 2010. Factorization machines. In *ICDM'10*. IEEE, 995–1000.

[28] Venu Satuluri and Srinivasan Parthasarathy. 2011. Symmetrizations for Clustering Directed Graphs. In *EDBT/ICDT '11*. 343–354.

[29] Venu Satuluri, Srinivasan Parthasarathy, and Yiye Ruan. 2011. Local Graph Sparsification for Scalable Clustering. In *SIGMOD '11*. 721–732.

[30] Sebastian Schelter, Venu Satuluri, and Reza Bosagh Zadeh. 2014. Factorbird - a Parameter Server Approach to Distributed Matrix Factorization. *ArXiv* abs/1411.0602 (2014).

[31] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-time Content Recommendations at Twitter. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1281–1292.

[32] Aneesh Sharma, C. Seshadhri, and Ashish Goel. 2017. When Hashes Met Wedges: A Distributed Algorithm for Finding High Similarity Vectors. In *WWW '17*. 431–440.

[33] Charalampos Tsourakakis. 2015. Provably Fast Inference of Latent Features from Networks: With Applications to Learning Social Circles and Multilabel Classification. In *WWW'15*. 1111–1121.

[34] Jaewon Yang and Jure Leskovec. 2013. Overlapping Community Detection at Scale: A Nonnegative Matrix Factorization Approach. In *WSDM'13*. 587–596.

[35] Jaewon Yang, Julian McAuley, and Jure Leskovec. 2014. Detecting Cohesive and 2-Mode Communities Indirected and Undirected Networks. In *WSDM'14*. 323–332.

[36] Xinyang Yi, Ji Yang, Lichan Hong, Derek Zhiyuan Cheng, Lukasz Heldt, Aditee Kumthekar, Zhe Zhao, Li Wei, and Ed Chi. 2019. Sampling-bias-corrected neural modeling for large corpus item recommendations. In *Recsys'19*. 269–277.

[37] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *KDD '18*. 974–983.

[38] Xiao Yu, Xiang Ren, Yizhou Sun, Quanquan Gu, Bradley Sturt, Urvashi Khandelwal, Brandon Norick, and Jiawei Han. 2014. Personalized entity recommendation: A heterogeneous information network approach. In *WSDM'14*. 283–292.

[39] Yongfeng Zhang, Qingyao Ai, Xu Chen, and W Bruce Croft. 2017. Joint representation learning for top-n recommendation with heterogeneous information sources. In *CIKM'17*. 1449–1458.

A SUPPLEMENT: FURTHER EVALUATION

The code for Neighborhood-aware MH and an in-memory implementation of Stage 1 are open-sourced in https://github.com/twitter/sbf.

A.1 Neighborhood-aware MH Empirical evaluation

A.1.1 Comparison with RandomMH [33]. We conducted a simple empirical evaluation in which we generated synthetic graphs with 100 nodes and varying number of communities, such that the probability of an edge between nodes inside the same community was large and the probability of an edge otherwise was small. The approach from [33] which we label 'RandomMH', as well as our approach ('Neighborhood-Aware') are implemented in the same code and use the same settings, except that the implementations for the proposal and the initialization functions are different. We compare both the approaches in terms of how many epochs they need to be to run to recover the synthetic communities, as well as the wall clock time (since the runtime for each epoch differs between the two approaches). As can be seen from the results in Table 2, the number of epochs and time required when using RandomMH grows exponentially with increasing k, as expected. Neighborhood-aware MH on the other hand has no such problem with increasing k.

A.1.2 Comparison on real datasets. We ran experiments on 8 real datasets (see Table 3) and compared Neighborhood-aware MH to the following algorithms from prior literature: (a) **BigClam** [34]: BigClam is interesting to compare to since there are many similarities, with the main difference being that it's optimized using gradient descent rather than randomized combinatorial optimization as in our case. We used the implementation in the SNAP package [21]. (b) **Graclus** [7]: Graclus optimizes weighted graph cuts without needing to compute eigenvectors, making it much faster than spectral algorithms without losing accuracy. [6] Note that for all 8 of these datasets, the RandomMH algorithm proposed in [33] was not able to make any progress inside the allotted time (6 hours).

We use two kinds of datasets: similarity graphs calculated for a subset of Twitter users in the way described in Section 3.1, as well as the 4 biggest undirected social networks we were able to find externally on the KONECT [19] collection. While our method (Neighborhood-Aware MH) and Graclus both work with weighted graphs, BigClam does not, so we restrict ourselves to unweighted graphs. For Neighborhood-aware MH, we run it with $l = 1$, i.e. each node gets assigned to at most one community, to keep the comparison with Graclus fair. α which can be used to trade precision with recall[7] was set to 10, and T, number of epochs, was set to 5. All experiments were run on a 16-core machine with 256GB RAM.

We evaluate all methods on **Precision** and **Recall**. A method is said to predict the existence of an edge (u, v) if u and v share at least one community per the output of the method. The *Precision* of a method is the proportion of actually existing predicted edges among all predicted edges for a method. The *Recall* of a method is the proportion of correctly predicted edges (by that method) among all actually existing edges in the graph. Given *Precision* and *Recall*, *F1* is their harmonic mean. Note that *Precision* and *Recall* are not properties of a community by itself, but rather are properties of the entire output i.e. the (possibly overlapping) set of communities. Note that our evaluation measures do not need any external ground-truth; they simply measure how well the community assignments are able to reconstruct the input graph.

Table 2: Epochs and time comparison for a synthetic graph with planted communities with 100 vertices and varying k

	RandomMH [33]		Neighborhood-aware MH	
k	Epochs	Time (seconds)	Epochs	Time (seconds)
2	30	0.09	3	0.02
4	200	0.19	5	0.02
8	100,000	23.4	3	0.03

For all of the datasets, we generally tried to set k – the number of discovered communities – so that the average size of a community is 100, because we see that having larger communities leads to significantly degraded *Precision* as unrelated pairs of nodes start to share at least one community. In the case of the `Orkut` and `Livejournal` datasets however, we used a smaller k in order to get at least one of our baselines to run successfully.

For BigClam, we found that the default implementation was taking a very long time (more than 100× the time for our method on our smallest dataset), so we made a modification to initialize using a random neighborhood (same as our method) instead of trying to identify the neighborhoods with the best conductance which was proving very expensive. Despite this optimization, BigClam was unable to finish execution within 6 hours for our 3 biggest datasets. For `Actors` and `Petster`, we found that BigClam finished execution successfully, but the results were completely unintelligible and seemed to have been affected by an unidentified bug.

As can be seen from the results in Table 3, our method is able to produce significantly more accurate results and is also much faster, typically 10x-100x faster. Neighborhood-aware MH is fast because each epoch requires making a single pass over all the vertices and their adjacency lists and also because the overall approach is easy to parallelize. Our approach is able to run inside 1.5 hours for a graph with 100M nodes and 5B edges (`Top100M`), while the largest graph either of our baselines is able to run on is at least an order of magnitude smaller.

A.2 Bipartite Communities Empirical evaluation

A possible concern with our approach to discovering bipartite communities is whether breaking the problem up into 3 separate steps can result in a loss of accuracy, as compared to jointly learning the bipartite communities directly. To understand this empirically, we compare against NMF (Non-negative Matrix Factorization) – recall that with both NMF and our approach, the end output is two low-dimensional sparse matrices. Specifically we use Scikit-Learn's implementation [3, 25] of alternating minimization using a Coordinate Descent solver, and with 'nndsvd' initialization, and with $L1$ penalty, where the $L1$ coefficient is adjusted to return results of comparable sparsity to our approach. For our approach, we set

[6] We downloaded it from http://www.cs.utexas.edu/users/dml/Software/graclus.html

[7] We find that it is easy to tune precision vs recall using α. Detailed results omitted.

Results on similarity graphs of Twitter users															
				Neighborhood-Aware MH (**Ours**)				BigClam				Graclus			
Dataset	$\|V\|$	$\|E\|$	$\|k\|$	Prec.	Rec.	F1	Time	Prec.	Rec.	F1	Time	Prec.	Rec.	F1	Time
Top100K	100K	6.5M	1,000	46.9	54.9	**50.6**	**< 1 min**	7.9	56.6	13.9	< 1 min	10.2	41.4	16.3	< 1 min
Top1M	1M	72M	10,000	44.8	42.5	**43.6**	**2 mins**	4.3	40.5	7.8	12 mins	7.9	30.5	12.6	16 mins
Top20M	20M	1.6B	200,000	34.7	45.0	**39.2**	**31 mins**	Did not finish in 6 hours				Fails with an error			
Top100M	100M	5.1B	500,000	31.1	46.8	**37.4**	**88 mins**	Did not finish in 6 hours				Fails with an error			
Results on the four biggest undirected social networks publicly available on KONECT [19]															
				Neighborhood-Aware MH (**Ours**)				BigClam				Graclus			
Dataset	$\|V\|$	$\|E\|$	$\|k\|$	Prec.	Rec.	F1	Time	Prec.	Rec.	F1	Time	Prec.	Rec.	F1	Time
Orkut	3M	117.2M	10,000	15.7	26.5	**19.7**	**2 mins**	Did not finish in 6 hours				1.7	26.2	3.2	1.9 hours
LiveJournal	5.2M	48.7M	10,000	17.4	27.6	**21.4**	**1 min**	Did not finish in 6 hours				0.5	46.6	1.1	4 hours
Actors	382K	15M	3,800	23.9	38.6	**29.5**	**< 1 min**	Fails with an error				4.4	37.8	7.8	4 mins
Petster	150K	5.4M	1,500	15.3	31.5	**20.6**	**< 1 min**	Fails with an error				1.0	8.3	1.8	3 mins

Table 3: Comparison with BigClam and Graclus for discovering communities from undirected graphs.

Results on directed sub-graphs of an undisclosed social network										
Dataset	$\|L\|$	$\|R\|$	$\|E\|$	k	Method	NNZ/row in **U**	NNZ/row in **V**	Correlation	AUC	Time
Employee	3,450	3,450	320K	34	**SimClusters**	10.4	1.3	**0.70**	0.91	< 1 min
					NMF	6.4	5.2	0.66	0.93	< 1 min
Verified20K	20K	20K	1.6M	200	**SimClusters**	14.7	1.8	**0.74**	0.95	3 mins
					NMF	9.5	5.3	0.65	0.95	10 mins
Results on directed/bi-partite graphs from KONECT [19]										
HepPH	32,158	28,230	421K	282	**SimClusters**	3.5	1.2	**0.72**	0.90	< 1 min
					NMF	3.0	1.8	0.64	0.89	3 mins
Reuters21K	19,757	38,677	980K	386	**SimClusters**	16.3	1.0	**0.68**	0.89	1 min
					NMF	25.4	0.6	0.40	0.88	3 mins

Table 4: Comparison against NMF for the usefulness of the learned U, V for link prediction.

various parameters as follows: for the similarity graph calculation in step 1, we only include edges with cosine similarity > 0.02; for Neighborhood-aware MH in step 2, we set $l = 4$, (i.e. each right-node can be assigned to at most 4 communities), α (see Eqn 1) to 10, and T (max epochs) to 5; for calculating **U** in step 3, we assign a left-node to a community if and only if it is connected to at least 2 right-nodes that are assigned to that community. All experiments were run on commodity servers with 8 cores and 24GB RAM. Note that this evaluation is purely to benchmark the accuracy of our approach; in terms of actual applicability, neither NMF nor other variants are practically feasible at our scale.

For evaluation, we use a combination of directed graphs and document-word occurrence graphs, and evaluate on the task of link prediction. We run both the approaches on 90% of the input dataset, and make a test set consisting of 10% of the held-out edges as well as the same number of randomly generated pairs of nodes which serve as negative examples in the test set. E.g., if a graph consists of $100K$ edges, this results in a "training set" of $90K$ edges and test-set of $20K$ edges ($10K$ positives and $10K$ negatives). In order to predict whether an edge (i, j) exists, we use the cosine similarity of $\mathbf{U}(i)$ and $\mathbf{V}(j)$ as the predicted score for the existence of the edge. (For both NMF and our approach, cosine similarity worked marginally better than dot product.) We evaluate the quality of these predicted scores in two ways - the first is we check the **Correlation** of the true label $\{0, 1\}$ with the predicted score; and the second is to calculate the **AUC** (Area Under the ROC Curve).[8]

Details about the datasets as well as the results are in Table 4. In terms of Correlation, our approach is consistently better across all datasets, while in terms of AUC, both the approaches are comparable. We also include the timing information, where our approach is generally a little faster than NMF. However, note that the primary advantage of our approach is not that it's faster than NMF, but that it's more *scalable*, meaning that it is possible to extend to billion-node graphs and hundreds of thousands of latent dimensions while scaling NMF similarly is prohibitively costly.

[8] Correlation with a binary variable is also known as point-biserial correlation and for a fixed dataset, it's proportional to the ratio of "difference in mean score for positive vs negative examples" to "the standard deviation of all the scores". See https://en.wikipedia.org/wiki/Point-biserial_correlation_coefficient.